The Specification and Verified Decomposition of System Requirements Using CSP

Andrew P. Moore
Code 5542, Naval Research Laboratory

Appears in IEEE Transactions on Software Engineering, Vol. 16, No. 9, Sept. 1990

Abstract - An important principle of building trustworthy systems is to rigorously analyze the critical requirements early in the development process, even before starting system design. Existing proof methods for systems of communicating processes focus on the bottom-up composition of component-level specifications into system-level specifications. Trustworthy system development requires, instead, the top-down derivation of component requirements from the critical system requirements. This paper describes a formal method for decomposing the requirements of a system into requirements of its component processes and a minimal, possibly empty, set of synchronization requirements. The Trace Model of Hoare's Communicating Sequential Processes (CSP) is the basis for the formal method. We apply the method to an abstract voice transmitter and describe the role that the EHDM verification system plays in the transmitter's decomposition. In combination with other verification techniques, we expect that the method defined here will promote the development of more trustworthy systems.

Index Terms - Automated theorem proving, communicating processes, formal specification, formal verification, process algebras, requirements definition, trusted systems, safety, security.

I. Introduction

A *critical system* is any system that can behave catastrophically. A *critical requirement* of a system is any requirement that if not satisfied can result in catastrophic behavior. In security-critical systems, a catastrophe might be the unauthorized disclosure of information; in safety-critical systems, it might be the uncontrolled release of energy. Most critical systems require an extremely high degree of assurance that they meet their critical requirements.[1] In combination with more conventional verification and validation techniques, formal methods can help attain this increased level of assurance.

An important step in applying formal methods to the development of a trustworthy system is to specify formally its critical requirements. Once these are specified and the design process has begun, methods are needed to derive lower-level component requirements from these critical requirements. Past work [1,2,3,4] suggests that process algebras, such as Hoare's Communicating Sequential Processes (CSP) [5], facilitate the formal specification of a system's critical requirements. Unfortunately, existing proof methods for these languages focus on the bottom-up composition of component-level specifications into system-level specifications, rather than a top-down derivation of component requirements from the (critical) system requirements. With this in mind, we define a formal method for

[1] Throughout this paper, a *system* is viewed as a network of communicating components (or component processes). A system is considered *trustworthy* if there is an acceptably high probability that it satisfies all its critical requirements.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1990**	2. REPORT TYPE	3. DATES COVERED **00-00-1990 to 00-00-1990**

4. TITLE AND SUBTITLE **The Specification and Verified Decomposition of System Requirements Using CSP**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5542,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**30**	

decomposing requirements of a system into requirements of its component processes and a minimal, possibly empty, set of synchronization requirements. CSP is the basis for describing the functionality of systems; the Trace Model is the basis for reasoning about properties of systems described in CSP[6,7].

The specification and decomposition of system requirements proceeds in successive stages of refinement and proof. Figure 1 illustrates the three layers of specification required for a decomposition. As shown in the highest layer, requirements may exist that cannot be partitioned completely into requirements on an individual component; such requirements involve the synchronized behavior of two or more components. Since these synchronization requirements are typically more difficult to verify, the decomposition method promotes reducing their number and complexity as far as possible. The set of these requirements is *minimal* if, when each requirement is described in conjunctive normal form, each conjunct of each requirement depends on the behavior of two or more components.

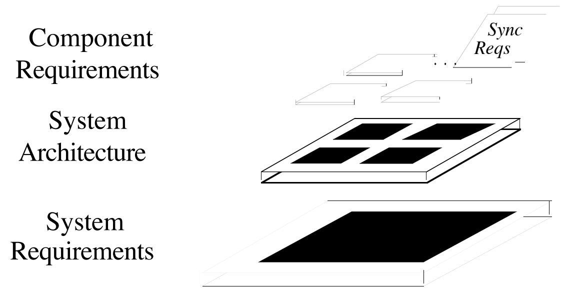

Figure 1. System Requirement Decomposition

Section II of this paper compares our method with other methods based on similar goals. Section III presents an overview of the relevant elements of the Trace Model of CSP, extends the CSP notation to facilitate hierarchical decomposition, and refines the method discussed above based on the model described. Section IV describes in detail the application of the method to an abstract voice transmitter. Finally, Section V discusses the use of the EHDM verification system [8,9] to check mechanically the integrity of the decomposition process, both in general and as it was used in the decomposition of the voice transmitter. General conclusions of this effort and plans for future research are also presented. Proofs in the main body of the paper are written in the style used in [10], where justifications for each proof step are provided as hints enclosed in curly brackets. The complete proofs of the theorems used in this paper and the decomposition of the voice transmitter using EHDM are documented in [11,12].

II. Comparison with Related Work

Hoare's work on CSP [5,13] and Milner's work on CCS [14,15] have provided the basis for many of the methods described for the design and verification of concurrent systems. Previous comparisons [6,16,17,18] characterize these methods by, among other things, the model of concurrency used, the types of properties provable, and the structure of systems specifiable. Due to the wealth of work done in this area we restrict our comparison primarily to methods based on CSP. Rather than explicitly describing each method and comparing it with our own, we describe how our method fits into the previously defined characterizations. This provides an implicit comparison with much of the previous work in a relatively short amount of space. We also compare our work with efforts to use traces for the abstract description of software and with more recent work not included in the past characterizations.

A. A Family of CSP Models

Olderog and Hoare [6] describe a family of increasingly sophisticated models for communicating processes: the Counter Model [19,20], the Trace Model [7,21,22], the Divergences Model, the Readiness Model [19,23], and the Failure Model [24,25,26]. Starting with the least sophisticated, the Counter Model involves the description of systems using separate channel histories. This model is shown to deal adequately only with acyclic or tree-like networks of processes. The Trace Model allows the description of arbitrary networks of processes. Both the Counter Model and the Trace Model can specify safety properties,[2] but neither can deal adequately with diverging processes.[3] The Divergences Model suffices to reason about systems that may diverge. The Readiness and Failure Models are required to reason about liveness in addition to safety. More recent work proposes the use of algebraic equations, rather than traces, to describe and prove properties about CSP [27].

Olderog and Hoare prove that the Trace Model is sufficient to reason about safety properties of non-divergent cyclic networks of processes. We choose this model for specifying and decomposing system requirements as a balance between its expressive power and the complexities involved with its use. For example, the Counter Model makes it difficult to specify certain relationships of values transmitted across different channels due to the fact that channel histories are recorded separately. The Gypsy Verification Environment [28,29] exhibits this same difficulty. Although useful for the verification of a variety of concurrent applications, Gypsy has a number of limitations on the form of the specification and implementation of concurrent programs. At the time this paper was written, it was very difficult to specify exit conditions of the form "at the time a message is received over channel A, the history of channel B (distinct from A) satisfies property P." The Trace Model alleviates this problem by interleaving the individual channel histories in the order in which the communications take place.

B. Characteristics of Trace-Driven Proof Systems

Hooman [17] and Barringer [18] describe a number of characteristics of proof systems for networks of processes. They distinguish between proof systems based on shared variables [17,30,31] versus those based on message passing, e.g., most systems based on CSP. Our method requires specifically that the only way one process may communicate with another process is through the transmission of messages over channels. Hooman distinguishes between proof systems based on a posteriori verification [32,33] versus verification as part of the design process [21,34,35]. Since the design and

[2] Safety for concurrent processes corresponds to partial correctness for sequential programs. Intuitively, safety properties specify that some condition does not occur whereas liveness properties specify that some condition will occur.

[3] A network of processes is *non-divergent* if all recursion is guarded and there is no possibility of the network engaging in an infinite consecutive sequence of hidden events. Note that while all terminating processes are non-divergent, not all non-divergent processes terminate.

implementation of nontrivial systems rarely, if ever, proceeds without error on the first attempt, an interactive verification/development process is an important part of our method.

A proof system is *compositional* if it is possible to derive a specification of a system from the specification of its components without knowledge of the component's internal structure. The verify-while-develop paradigm of system development has led to the need for compositional proof systems [21,30,34,35] versus non-compositional proof systems [31,33]. We introduce a special compose operator to CSP allowing the description of a process based solely on the external channels over which the process communicates. By adapting proof rules from [5], a compositional proof method is defined similar to the approach used by van de Snepscheut for the verification of hardware designs [10]. However, unlike previous work, our method emphasizes the decomposition of high-level requirements, rather than the composition of low-level specifications.

In a slightly different vein, Bartussek and Parnas [36] and, more recently, Parnas and Wang [37] describe the use of trace theory for the abstract specification of software modules. Rather than reasoning about traces of communications, as does the Trace Model of CSP, this work reasons about software using traces of procedure calls. McLean extended the original work by providing a formal foundation for specifying software requirements using traces [38] and defining the trace semantics of a simple sequential programming language [39]. This provides a general framework for specifying and verifying sequential program behavior. In combination with the method described in this paper for decomposing system-level requirements into component-level requirements, this framework may be useful for specifying and verifying the critical requirements of concurrent software systems.

C. Applications to Critical Systems

McCullough [40] and Jacob [1] use trace-based methods to reason about the multi-level security requirements of systems. McCullough introduces the notion of a composable property as one that if proven of the components of a systems, is true of the system as a whole. He derives a composable security property, called restrictiveness, and demonstrates its use in constructing multi-level secure systems. Unfortunately, many critical requirements of interest may not be composable and, therefore, are not subject to the methods McCullough proposes. Composability applies only to systems composed of components similar enough that the same critical requirements apply to all components and to the system as a whole. The goal of our method is to tackle the more general problem of deriving component requirements from arbitrary system requirements.

Jacob [1] demonstrates the expressive power of the Trace Model of CSP by specifying complex multi-level security requirements of systems. In later work, Woodcock [41] and Jacob [42] describe a method to derive an implementation from a specification not by intelligently guessing, but rather under guidance from the structure of the specification. Woodcock's approach requires stating a set of fairly low-level requirements and generating an implementation consisting of a parallel composition of processes, each component of which implements a requirement. Jacob discusses problems with a (possibly nonterminating) method of generating more secure designs from less secure designs using Woodcock's basic approach. Our method is different in that we are trying to derive low-level requirements from a given high-level requirement and (partial) implementation. We do not assume that we have the freedom to choose the implementation since that choice may depend on other considerations, e.g., the required non-critical functionality of the system or the physical realization of the system in hardware.

III. The Decomposition Method

This section describes an iterative seven-step method for decomposing system requirements.[4] We present an overview of relevant elements of the CSP Trace Model and describe a (primarily syntactic) extension for constructing systems in a hierarchical manner. In the course of this discussion, three rules are introduced that are used to justify the definition of the method. We use standard notation where possible and describe CSP-specific notation when first used. We assume universal quantification of variables in formulas unless otherwise indicated. Readers interested in a summary of the notation used or other details of CSP not covered in this paper should refer to [5].

A. CSP Preliminaries

A CSP process communicates with its environment through named communication channels. The Trace Model of CSP maps a process to an alphabet and a set of traces. The alphabet of a process P, denoted αP, specifies all events in which P is permitted to engage. A trace of a process is an observation of its execution. It consists of a finite sequence of events in which the process has engaged at some moment in time. The set of all traces of a process P, denoted traces(P), is a prefix-closed nonempty subset of αP^*, where αP^* is the set of all finite traces formed from events in αP.[5]

The CSP notation allows the description of processes using a variety of process constructors. This paper primarily deals with the commutative concurrency operator, $\|$. $P \| Q$ describes a process executing process P concurrently with process Q. $P \| Q$ requires P and Q to participate simultaneously in those events that occur in both αP and αQ. Events occurring in αP but not αQ may be engaged in by P independently of Q. Since either process of a concurrent composition may itself be a concurrent process, this operator supports the description of arbitrary networks of processes.

Processes executing concurrently communicate through channels. A communication event is a special type of event described by a pair $c.v$. The alphabet of process P contains $c.v$ if and only if P is permitted to communicate message v over channel c. $c\ !\ v$ denotes the output of value v on the channel c; $c\ ?\ m$ denotes the input of any value m communicable on the channel c. These operations are communication events defined by

Definition 1: $(c\ !\ v \rightarrow P) = (c.v \rightarrow P)$

Definition 2: $(c\ ?\ m \rightarrow P(m)) = (c.n{:}\alpha P \rightarrow P(n)))$

where the prefix process $e \rightarrow P$ describes a process that first engages in the event e and then behaves like process P; and the choice process $x{:}A \rightarrow P(x)$ describes a process that chooses x from A then behaves like $P(x)$. Although the CSP notation distinguishes between the input

4 Henceforth, the term *requirement* refers to the statement of a property. The term *specification* refers to the statement that a system, or process, satisfies some property.

5 Traces(P) must be prefix-closed since every prefix of an observation of P must also be an observation of P. The set is nonempty since the empty trace is a valid observation of every process.

and output of values over channels, the Trace Model uses only the generic dot notation, $c.v$, to represent communications over channels. For example,

$$\text{traces}(c\,!\,v \rightarrow P) = \{<>\} \cup \{<c.v>\,\hat{}\,t \mid t \in \text{traces}(P)\}$$

where ^ is the trace append operation.

A communication of message *m* over channel *c* can occur between two processes running concurrently if and only if both processes have the communication event *c.m* in their alphabets and both processes simultaneously engage in that event. That is, whenever one process outputs a value onto the channel, the other process simultaneously inputs the same value from the channel.[6] This implies that

Definition 3: $((c\,!\,v \rightarrow P) \parallel (c\,?\,m \rightarrow Q(m))) = (c.v \rightarrow P \parallel Q(v))$

where $c.v$ occurs in αP and αQ. If only one process in a system has a communication event in its alphabet, then that process may engage in that event independently of any other process. To simplify the theory involved, Hoare assumes that at most two processes in a system can access the same communication channel and that communication over a channel occurs in only one direction. If only one process within the system can access the channel, the channel is said to be *external*; if two processes can access the channel, the channel is said to be *internal.*

A requirement in CSP is viewed as a set of traces. Process *P* satisfies a requirement *R,* denoted *P* sat *R,* if and only if *R* contains every trace that may occur as an observation of *P*:

Definition 4: $(P \text{ sat } R) = (\text{traces}(P) \subseteq R)$.

Clearly, a requirement is satisfiable by some process only if it contains a non-empty prefix-closed subset. For convenience we define a functional notation for requirements

Definition 5: $R(tr1) = (tr1 \in R)$

and a predicate ValidTrace as

Definition 6: $\text{ValidTrace}(tr1, P) = (tr1 \in \text{traces}(P))$.

Then, trivially,

Lemma 1: $(P \text{ sat } R)$ iff $\forall tr1.\ (\text{ValidTrace}(tr1, P) \Rightarrow R(tr1))$.

B. Concurrent Processes and Hiding

The Trace Model requires relating each process to an alphabet and a set of traces. Since our goal is to decompose the requirements of a system as far as possible into requirements of its components, it is helpful to define the alphabet and set of traces of a concurrent process in terms of its component processes. Clearly,

Definition 7: $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$.

The valid traces of a concurrent process are defined as

6 To account for the time it takes to transmit information over physical wires we can associate with each interface between two processes an additional process that delays transmissions for some length of time. For ease of exposition, this paper does not consider issues regarding transmission delay; for details see [12].

Definition 8: $\mathrm{ValidTrace}(tr1, P \parallel Q) = (\mathrm{ValidTrace}(tr1 \upharpoonright \alpha P, P)$
$\quad\wedge \mathrm{ValidTrace}(tr1 \upharpoonright \alpha Q, Q)$
$\quad\wedge tr1 \in (\alpha P \cup \alpha Q)^*)$,

where the restriction operator, $\upharpoonright$, takes a trace and a set of events and returns the trace with the elements not in the set removed. Hoare justifies Definition 8 by arguing that if *tr1* $\in$ traces($P \parallel Q$), then every event of *tr1* must be an element of either αP or αQ. For every event *e* in *tr1,* $e \in \alpha P$ if and only if *e* occurs in the trace of *P*. Likewise, for every event *e* in *tr1,* $e \in \alpha Q$ if and only if *e* occurs in the trace of *Q*. Therefore, $(tr1 \upharpoonright \alpha P) \in$ traces(P) and $(tr1 \upharpoonright \alpha Q) \in$ traces(Q). Definition 7 suggests that *tr1* must be an element of $(\alpha P \cup \alpha Q)^*$.

Definition 8 requires that any trace of a concurrent process $P \parallel Q$ include every event in which *P* or *Q* engage. The visibility of the communications over internal channels in the traces of $P \parallel Q$ reduces the amount of abstraction possible during the system design process. Hierarchical design, a proven method for managing the complexity of system design and verification, requires that the specification of a component be based solely on the sequence of external communications in which it may engage. To support this, we define the compose operator, denoted $|\backslash|$, equivalent to the CSP concurrency operator except that the internal communications are hidden; consequently,

Definition 9: $\alpha(P |\backslash| Q) = (\alpha P \div \alpha Q)$

where $(\alpha P \div \alpha Q) = (\alpha P \cup \alpha Q) - (\alpha P \cap \alpha Q)$. Further, we define a new operator, $\uparrow$, which extends the definition of $\upharpoonright$ to operate on sets of traces, as

Definition 10: $A \uparrow S = \{t \mid \exists t'. (t' \in A) \wedge (t' \upharpoonright S = t)\}$.

This extension allows the definition of the traces of $P |\backslash| Q$ in terms of the traces of $P \parallel Q$ as

Definition 11: $\mathrm{traces}(P |\backslash| Q) = (\mathrm{traces}(P \parallel Q) \uparrow (\alpha P \div \alpha Q))$.

Concealing communication over internal channels in this way is equivalent to that accomplished by the trace blend operation defined in [10].

Clearly from the above definitions, the alphabet and traces of a compose process contain only the external communication events in which the process may engage. Thus, each valid trace of a compose process corresponds to some valid trace of the related concurrent process as follows:

Rule 1: $\mathrm{ValidTrace}(tr1, P |\backslash| Q)$
$\quad\Rightarrow \exists tr2. (\mathrm{ValidTrace}(tr2, P \parallel Q)$
$\quad\quad\wedge tr1 = (tr2 \upharpoonright (\alpha P \div \alpha Q)))$

Proof:

$\mathrm{ValidTrace}(tr1, P |\backslash| Q)$

= {Definition 6}

$tr1 \in \mathrm{traces}(P |\backslash| Q)$

= {Definition 11}

$tr1 \in \mathrm{traces}(P \parallel Q) \uparrow (\alpha P \div \alpha Q)$

= {Definition 10}

$tr1 \in \{t \mid \exists\, tr2. (tr2 \in \mathrm{traces}(P \parallel Q) \wedge t = (tr2 \upharpoonright (\alpha P \div \alpha Q)))\}$

$\rightarrow$ **{Definition 6}**

$\exists\, tr2.\ (ValidTrace(tr2, P \parallel Q) \wedge tr1 = (tr2 \upharpoonright (\alpha P \div \alpha Q)))$

End of Proof.

Note that the compose operator is just a convenient notation for what could be defined using concealment in Hoare's CSP. For example,

$$(P \,|\backslash|\, Q) = ((P \parallel Q) \setminus (\alpha P \cap \alpha Q))$$

where the CSP concealment operator, \, is used to hide all communications over internal channels.

As mentioned previously, use of the Trace Model requires showing that the application of interest is non-divergent. Divergence arises from unguarded recursion. A recursion of the form $\mu P.F(P)$ is guarded if $F(P)$ starts with at least one event prefixed to all recursive occurrences of P. Divergence can also arise from the hiding of events as is accomplished through the use of the compose operator. Any process that can engage in an infinite consecutive sequence of hidden events is divergent. In the case of a compose process, divergence leads to infinite internal chatter. Hoare defines a theory for reasoning about divergent processes. From this theory we have defined a method for stating requirements on component processes sufficient to guarantee non-divergence of a system [12]. This is a fairly mechanical process and is performed in isolation from the requirement decomposition process; we, therefore, do not further consider issues of divergence in this paper.

C. Decomposing System Requirements

We are now able to state and prove the primary inference rules for decomposing requirements of a compose process. The first reduces the problem of proving requirements of a compose process to proving requirements of a concurrent process:

Rule 2: $((P \parallel Q)\ sat\ R$

$\quad \wedge\ \forall\, tr1.\ (ValidTrace(tr1, P \,|\backslash|\, Q)$

$\qquad \Rightarrow \exists\, tr2.\ (ValidTrace(tr2, P \parallel Q)$

$\qquad\quad \wedge\ (tr1 = tr2 \upharpoonright (\alpha P \div \alpha Q))$

$\qquad\quad \wedge\ (R(tr2) \Rightarrow R(tr2 \upharpoonright (\alpha P \div \alpha Q))))))$

$\quad \Rightarrow (P \,|\backslash|\, Q)\ sat\ R$

Proof:

$(P \parallel Q)\ sat\ R$

$\wedge\ \forall\, tr1.\ (ValidTrace(tr1, P \,|\backslash|\, Q)$

$\qquad \Rightarrow \exists\, tr2.\ (ValidTrace(tr2, P \parallel Q)$

$\qquad\quad \wedge\ (tr1 = tr2 \upharpoonright (\alpha P \div \alpha Q))$

$\qquad\quad \wedge\ (R(tr2) \Rightarrow R(tr2 \upharpoonright (\alpha P \div \alpha Q)))))$

$\rightarrow$ **{Lemma 1}**

$\forall\, tr1.\ (ValidTrace(tr1, P \,|\backslash|\, Q)$

$\qquad \Rightarrow \exists\, tr2.\ (R(tr2) \wedge (tr1 = tr2 \upharpoonright (\alpha P \div \alpha Q))$

$\qquad\quad \wedge\ (R(tr2) \Rightarrow R(tr2 \upharpoonright (\alpha P \div \alpha Q)))))$

= {Lemma 1}

$(P \,|\backslash|\, Q)\ sat\ R$

End of Proof.

The hypothesis of this rule requires proving properties about the requirements of a concurrent process. The following inference rule reduces the problem of proving requirements of a concurrent process to proving requirements of its components:

Rule 3: $((P \text{ sat } S) \wedge (Q \text{ sat } R)$

$\wedge\ \forall\, tr1.\ (ValidTrace(tr1,\ P \parallel Q)$

$\Rightarrow ((S(tr1 \upharpoonright \alpha P) \Rightarrow S(tr1)) \wedge (R(tr1 \upharpoonright \alpha Q) \Rightarrow R(tr1))$

$\wedge ((S(tr1) \wedge R(tr1)) \Rightarrow T(tr1)))))$

$\Rightarrow (P \parallel Q) \text{ sat } T$

Proof:

$(P \text{ sat } S) \wedge (Q \text{ sat } R)$

$\wedge\ \forall\, tr1.\ (ValidTrace(tr1,\ P \parallel Q)$

$\Rightarrow ((S(tr1 \upharpoonright \alpha P\ ) \Rightarrow S(tr1)) \wedge (R(tr1 \upharpoonright \alpha Q\ ) \Rightarrow R(tr1))$

$\wedge ((S(tr1) \wedge R(tr1)) \Rightarrow T(tr1))))$

$\rightarrow$ **{Definition 8}**

$\forall\, tr1.\ ValidTrace(tr1,\ P \parallel Q)$

$\Rightarrow (ValidTrace\ (tr1 \upharpoonright \alpha P,\ P) \wedge ValidTrace(tr1 \upharpoonright \alpha Q,\ Q)$

$\wedge (P \text{ sat } S) \wedge (Q \text{ sat } R) \wedge (S(tr1 \upharpoonright \alpha P\ ) \Rightarrow S(tr1))$

$\wedge (R(tr1 \upharpoonright \alpha Q\ ) \Rightarrow R(tr1)) \wedge ((S(tr1) \wedge R(tr1)) \Rightarrow T(tr1)))$

$\rightarrow$ **{Lemma 1}**

$\forall\, tr1.\ ValidTrace(tr1,\ P \parallel Q)$

$\Rightarrow (S(tr1 \upharpoonright \alpha P\ ) \wedge R(tr1 \upharpoonright \alpha Q) \wedge (S(tr1 \upharpoonright \alpha P\ ) \Rightarrow S(tr1))$

$\wedge (R(tr1 \upharpoonright \alpha Q\ ) \Rightarrow R(tr1)) \wedge ((S(tr1) \wedge R(tr1)) \Rightarrow T(tr1)))$

= {Lemma 1}

$(P \parallel Q) \text{ sat } T$

End of Proof.

Rules 1, 2, and 3 form the basis of our method for decomposing system requirements. Although relatively easy to prove, they encompass the conditions sufficient for justifying the method. The following describes and justifies the method using these rules:

1. *Describe the architecture of the system as a composition of processes that can be arranged in a binary tree $P_{i,j}$, for $0 \le i < n$ and $0 \le j < 2^{n-1}$. Define the alphabet of the system as the set of external communication channels, and only those communication channels, over which the system is permitted to communicate.*

This step requires describing, as a binary tree, that part of the system architecture of interest.[7] Each non-leaf vertex of the tree is a process representing the composition of its left son and its right son. Therefore, each subtree represents a subsystem of the entire system. The root of the tree, $P_{0,0}$, represents the system as a whole. The tree need not be complete,[8] but if a vertex has one son then it must have both sons. Later refinement of the architecture

[7] Arranging the processes in the form of a binary tree is done solely for the purpose of easing the statement of the method. This method is applicable to any network of communicating processes that is constructed with unidirectional two-process channels.

[8] The binary tree $P_{i,j}$ is *complete* if every vertex of depth less than n-1 has both a left son and a right son, and every vertex of depth n-1 is a leaf. The depth of a vertex v is the length of the path from the root to v.

specified in this step requires the straightforward application of Steps 1 through 7 to the new part of the architecture.

2. *Specify the necessary requirements of the system in the form $P_{0,0}$ sat $R_{0,0}$.*

The system specification is stated as a requirement $R_{0,0}$ of $P_{0,0}$. Subsequent decomposition will result in a requirement $R_{i,j}$ for each process $P_{i,j}$ of the system.

→ For $0 \leq i < n$ and $0 \leq j < 2^{n-1}$, let $SR_{i,j}$ be the derived synchronization requirement for $P_{i,j}$, defined as true initially.[9] Traverse the tree in a breadth-first manner. At each non-leaf vertex $P_{i,j}$, perform the following steps:

3. *Define the alphabets of $P_{i+1,2j}$ and $P_{i+1,2j+1}$. Reduce the specification of the compose process $P_{i,j}$ to the specification of the concurrent process ($P_{i+1,2j} \parallel P_{i+1,2j+1}$) by proving the* ***Compose Restriction Condition,***

$$R_{i,j}(tr1) \Rightarrow R_{i,j}(tr1 \upharpoonright \alpha P_{i,j}).$$

Definition 9 requires that the alphabets of the sons of $P_{i,j}$, $P_{i+1,2j}$ and $P_{i+1,2j+1}$,be defined such that their symmetric set difference equals the alphabet of $P_{i,j}$. Under these restrictions, $P_{i,j}$ sat $R_{i,j}$ follows from the conditions

$$\begin{array}{l}
1.\ (P_{i+1,2j} \parallel P_{i+1,2j+1}) \text{ sat } R_{i,j} \\
2.\ \forall\, tr1.\ \text{ValidTrace}(tr1,\ P_{i+1,2j}\ |\backslash|\ P_{i+1,2j+1}) \\
\qquad \Rightarrow \exists tr2.(\text{ValidTrace}(tr2,\ P_{i+1,2j} \parallel P_{i+1,2j+1}) \\
\qquad\qquad \wedge\ tr1 = (tr2 \upharpoonright (\alpha P_{i+1,2j} \div \alpha P_{i+1,2j+1})) \\
\qquad\qquad \wedge\ (R_{i,j}(tr2) \Rightarrow R_{i,j}(tr2 \upharpoonright (\alpha P_{i+1,2j} \div \alpha P_{i+1,2j+1}))))
\end{array}$$

by Rule 2. The second condition follows from Rule 1 and the proof of the Compose Restriction Condition. The first will be decomposed in subsequent steps. In this, and subsequent steps, we require proof of stronger conditions than necessary, e.g., the Compose Restriction Condition, so as to proceed without any specific knowledge about the valid traces of the concurrent process $P_{i+1,2j} \parallel P_{i+1,2j+1}$. Requirements that depend on such knowledge, referred to as synchronization requirements, will be included in $SR_{i,j}$ in Step 7.

4. *Derive requirements $R_{i+1,2j}$ for $P_{i+1,2j}$ and $R_{i+1,2j+1}$ for $P_{i+1,2j+1}$ with the goal of proving $P_{i+1,2j} \parallel P_{i+1,2j+1}$ satisfies $R_{i,j}$.*

This is the first attempt to determine requirements for the components $P_{i+1,2j}$ and $P_{i+1,2j+1}$. This is the most important, and often the most difficult, step in the decomposition process; the better the determination made here, the less work that is required in subsequent steps. It is important to make the requirements as weak as possible. Although finding the weakest requirements may be difficult, the weaker the requirements found, the less that will be required of $P_{i+1,2j}$ and $P_{i+1,2j+1}$ to satisfy $R_{i,j}$. This maximizes the amount of freedom in the design and implementation of the components while still meeting the system-level requirements defined.

5. *Prove the* ***Concurrent Restriction Condition,***

[9] At the completion of this method the combination of the synchronization requirements $SR_{i,j}$ forms the minimal set of synchronization requirements discussed in the introduction.

$$R_{i+1,2j}(tr1 \upharpoonright \alpha P_{i+1,2j}) \Rightarrow R_{i+1,2j}(tr1)$$
$$\wedge\ R_{i+1,2j+1}(tr1 \upharpoonright \alpha P_{i+1,2j+1}) \Rightarrow R_{i+1,2j+1}(tr1).$$

If this proof fails, revise the requirements so that $R_{i+1,2j}$ depends only on the events in the alphabet of $P_{i+1,2j}$ and that $R_{i+1,2j+1}$ depends only on events in the alphabet of $P_{i+1,2j+1}$ and try again.

Once the component requirements are formulated, application of Rule 3 requires proving the condition

$$\mathrm{ValidTrace}(tr1, P_{i+1,2j} \parallel P_{i+1,2j+1})$$
$$\Rightarrow (R_{i+1,2j}(tr1 \upharpoonright \alpha P_{i+1,2j}) \Rightarrow R_{i+1,2j}(tr1)$$
$$\wedge\ R_{i+1,2j+1}(tr1 \upharpoonright \alpha P_{i+1,2j+1}) \Rightarrow R_{i+1,2j+1}(tr1)).$$

Proof of the Concurrent Restriction Condition is sufficient, though not necessary, to prove this requirement. If it cannot be proven, $R_{i+1,2j}$ must be restricted so as to be independent of the events outside of $\alpha P_{i+1,2j}$, and $R_{i+1,2j+1}$ must be restricted so as to be independent of the events outside of $\alpha P_{i+1,2j+1}$.

6. *Attempt to prove the **Conjunction Condition,***

$$(R_{i+1,2j}(tr1) \wedge R_{i+1,2j+1}(tr1)) \Rightarrow R_{i,j}(tr1).$$

If successful, continue the tree traversal to the next vertex at step 3. Otherwise, specify the weakest condition, C, needed to complete the proof.

Application of Rule 3 requires proving that

$$\mathrm{ValidTrace}(tr1, P_{i+1,2j} \parallel P_{i+1,2j+1})$$
$$\Rightarrow ((R_{i+1,2j}(tr1) \wedge R_{i+1,2j+1}(tr1)) \Rightarrow R_{i,j}(tr1)).$$

Proof of the Conjunction Condition is sufficient to satisfy this requirement. If it is provable, the decomposition at the current vertex is complete; in this case, $P_{i,j}$ sat $R_{i,j}$ provided that $P_{i+1,2j}$ sat $R_{i+1,2j}$ and $P_{i+1,2j+1}$ sat $R_{i+1,2j+1}$. Otherwise, Step 7 requires deriving a condition that allows completion of the proof. As in step 4, the weaker the condition found, the less that will be required of $P_{i+1,2j}$ and $P_{i+1,2j+1}$ to satisfy $R_{i,j}$. The weakest condition must be found in order to guarantee the minimization of the set of synchronization requirements.

7. *Describe C as a conjunction of simple conditions in conjunctive normal form. If no conjunct depends solely on the traces of either $P_{i+1,2j}$ or $P_{i+1,2j+1}$ then conjoin C to $SR_{i,j}$ and continue the tree traversal to the next vertex at step 3. Otherwise, conjoin to $R_{i+1,k}$ each conjunct of C that depends only on the traces of $P_{i+1,k}$ (for $k = 2j$ or $k = 2j+1$) and continue at step 5.*

Requirements that depend solely on the behavior of one component process are integrated into that process's specification whenever possible. Describing *C* as a conjunction of conditions, each captured in the simplest possible context, helps ensure that no requirement exists that could be stated as part of the specification of one of the component processes. This, in turn, maximizes the benefit gained from the decomposition process by minimizing the number and complexity of the synchronization requirements. Those requirements that depend on two or more component processes depend on specific knowledge about the valid traces of the concurrent process and are added to $SR_{i,j}$.

D. Assumptions

The above method reduces the problem of formally verifying the requirements of a concurrent system into two separate, simpler problems: verifying that the system components meet their derived requirements and verifying that specific combinations of those components meet any derived synchronization requirements. Of course, the eventual implementation of the components of the system and their interconnections must be shown to satisfy the assumptions of the Trace Model of CSP on which our method depends. In summary, these are as follows:

1. The only way a process can communicate with another process executing concurrently is through CSP-like communication channels; no shared variables are permitted.
2. Exactly those external communication channels over which a process may pass data are included in its alphabet.
3. At most two processes of a system may communicate over a given channel; no broadcast capability exists. If the channel is external, exactly one process in the system must have the communication events associated with that channel in its alphabet. If the channel is internal, exactly two processes must have the communication events associated with that channel in their alphabets.
4. Communication over a given channel may take place in one direction only.
5. Any assumptions made of the communication media, e.g., in the definition of a transmission delay, must be validated.

Assuring the validity of these assumptions must take place throughout system development.

IV. An Example Decomposition

This section presents a description of the verified decomposition of an abstract voice transmitter using, as a guide, the iterative seven-step method discussed in Section III. The example, called the μASVT, is a simplified version of a voice terminal specified previously [2]. Although the notation $P_{i,j}$ for processes, $R_{i,j}$ for component requirements, and $SR_{i,j}$ for synchronization requirements is convenient for describing the method for an arbitrary application, for a specific application this notation is cumbersome. Throughout the specification and decomposition of the μASVT, we use mnemonic names for processes so that, for example, the requirements for the process μASVT are $R_{\mu ASVT}$ and $SR_{\mu ASVT}$.

A. The μASVT: An Informal Description

The μASVT allows for the encrypted or plain text transmission of voice. It consists of three major components - the Voice Processor, the Modem Processor, and the Comsec Module. Figure 2 illustrates the two external interfaces, Red Channel and Black Channel, and the three internal interfaces, Voice Modem, Voice Comsec, and Modem Comsec, through which voice transmissions may flow. The Red and the Black Channels are both analog interfaces. Conceptually, the Red Channel may be connected to telephone sets or intercoms; the Black Channel may be connected to radios or wireline appliques.

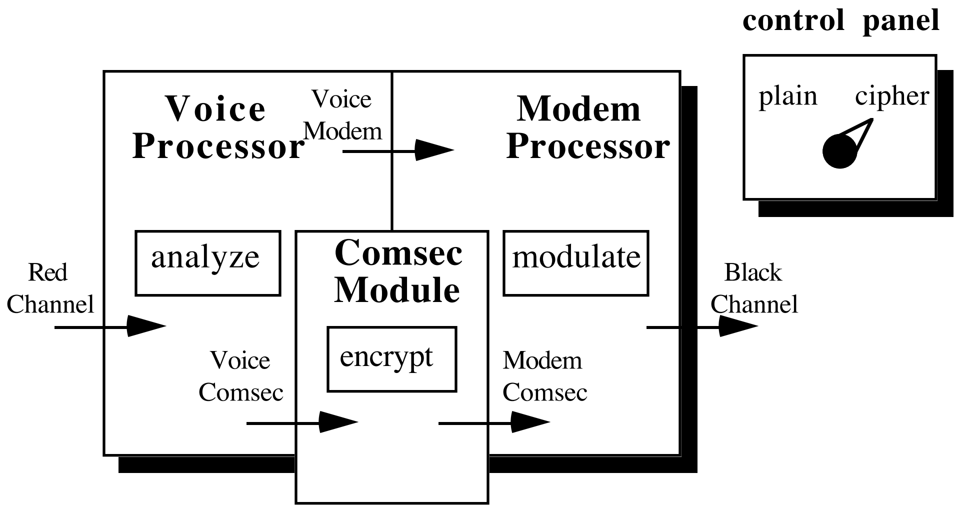

Figure 2. The μASVT

The μASVT has a control panel allowing its users to choose between the cipher or the plain mode of operation. When in the cipher mode, the Voice Processor analyzes outgoing transmissions, the Comsec Module encrypts these transmissions, and the Modem Processor codes important bits and modulates the resulting bit stream. Transmissions in the plain mode are processed similarly except that the Comsec Module is bypassed so that information is transmitted through the Voice Modem channel without being encrypted. The terminal must be clear of all voice transmissions before a user may change the status of the control panel. The key used for encryption is pre-defined by the system and never changes. A user can receive transmissions from the μASVT only if he has access to a receiver that inverts the transformation performed by the terminal. This, of course, requires access to the decryption key. The details of the μASVT transformations have little or no bearing on the statement of its requirements and for ease of exposition are omitted.

The μASVT has one critical requirement, Red/Black Separation. More specifically, all information transmitted by the terminal when in the cipher mode of operation must be Black, encrypted, data. Thus, the only way for the μASVT to transmit Red, plain text, data is when the control panel is set in the plain mode. Because of its restricted functionality, the μASVT serves only as a demonstration of the method discussed in this paper - it is not intended as a real-world system. The practicality of this method and its demonstration on real systems is a topic of future research.

B. Specifying μASVT Architecture and Requirements

1. *Describe the architecture of the μASVT as a composition of processes that can be arranged in a binary tree $P_{i,j}$, for $0 \leq i < n$ and $0 \leq j < 2^{n-1}$. Define the alphabet of the μASVT as the set of external communication channels, and only those communication channels, over which the μASVT is permitted to communicate.*

The μASVT is described as the composition of three CSP processes, VP representing the Voice Processor, CM representing the Comsec Module, and MP representing the Modem Processor. Let μASVT be the CSP process representing the terminal as a whole. Then,

Definition 12: μASVT = (VP |\| (CM |\| MP)).

Figure 3 illustrates the μASVT architecture as a binary tree of processes. This view requires that we introduce a new process name CMP, for Comsec Modem Processor, representing the composition of CM and MP.

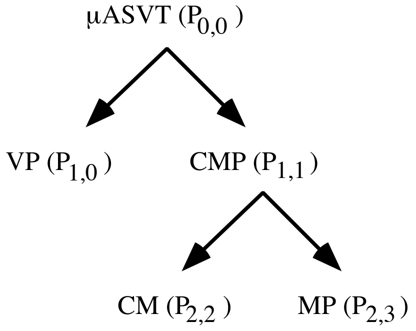

Figure 3. μASVT Architecture

Define M to be the set of all possible messages communicable over the message channels. The alphabet of the μASVT is defined to be those communication events in which it is permitted to engage at its external interface:

Definition 13: αμASVT = {VPCtl.c, RedChan.m, BlackChan.m | c∈{cipher, plain}, m∈M}.

The message channels referenced here correspond to the external channels described in the informal description. VPCtl is a control channel that notifies the μASVT of an input from the control panel; these inputs can be either cipher or plain signifying whether the terminal is in cipher mode or plain mode.

2. *Specify the necessary requirements of the system in the form μASVT sat $R_{\mu ASVT}$.*

Specifying the μASVT's critical requirement, Red/Black Separation, requires that every message transmitted while in the cipher mode of operation be encrypted.[10] The Restriction operator, ↾, provides a mechanism for specifying properties about the values transmitted over particular channels in isolation from other communications. Specifying Red/Black separation requires a mechanism stronger than this to determine, given an arbitrary trace, the sequence of values communicated over a channel when in the cipher mode of operation. We define an abstract mechanism for doing this since it will be useful in the specification of the components as well as the μASVT. The mechanism is based on a function Filter,

[10] Note that no attempt is made to formally specify or decompose the non-critical functionality requirements in this paper.

```
Definition 14: Filter(tr1,set1,req1)
                 = if tr1 = < > then < >
                   elsif Last(tr1) ∈ set1 ∧ req1(NonLast(tr1))
                    then Append(Last(tr1),
                                  Filter(NonLast(tr1),set1,req1))
                    else Filter(NonLast(tr1),set1,req1),
```

where < > denotes the empty sequence, Last returns the last element of a sequence, NonLast returns all but the last element of a sequence, and Append appends an element to the end of a sequence. Intuitively, Filter returns the sequence of elements of trace *tr1* that are in set *set1* such that the sequence of events leading up to each element satisfies requirement *req1*.

To use Filter, we define a function, CipherMode, that determines whether the terminal is in the cipher mode after an arbitrary sequence of communications takes place,

```
Definition 15: CipherMode(ctlch1)(tr1)
                 = if tr1 = < > then (InitialMode = cipher)
                   elsif Last(tr1) = ctlch1.c then (c=cipher)
                   else CipherMode(ctlch1)(NonLast(tr1)),
```

where InitialMode is the position of the control panel on start-up and *ctlch1* is the control channel over which the cipher/plain mode signals are sent. Now, if *ch1*:M denotes the set of all communications of values in M over channel *ch1*,

```
Filter(tr1, ch1:M, CipherMode(ctlch1))
```

returns the sequence of communications over *ch1* that occur in *tr1* either (1) when the last value sent over control channel *ctlch1* was cipher, or (2) if there is no such value, when InitialMode is cipher.

Red/Black separation requires that every message output over BlackChan correspond to the proper transformation, including an encryption, of a message input over RedChan. Let Key be the key used by the μASVT for encryption of voice and the functions Analyze, Encrypt, and Modulate be the functions representing the transformations performed by the Voice Processor, Comsec Module, and Modem Processor, respectively. Using the Filter function as described above, Red/Black separation is captured in the specification of the μASVT:

Specification 1: μASVT sat $R_{\mu ASVT}$

```
R_μASVT(tr1)
  = ∀ m1.(BlackChan.m1 in Filter(tr1,BlackChan:M,
                                  CipherMode(VPCtl))
          ⇒ ∃ m2.(RedChan.m2  in Filter(tr1,RedChan:M,
                                           CipherMode(VPCtl))
                  ∧ Modulate(Encrypt(Analyze(m2),Key)) = m1)),
```

where **in** is the sequence membership symbol.

C. Decomposing μASVT Requirements: The First Attempt

The top level CSP specification of the μASVT composes VP with (CM |\| MP). Although a full analysis requires a complete traversal of the tree illustrated in Figure 3, the following describes only the decomposition of μASVT into two components, VP and CMP, where

Definition 16: CMP = (CM |\| MP);

the decomposition of CMP proceeds similarly. Section IV-E describes the final results of the full decomposition of the μASVT. The appendix to this paper presents an example CSP implementation of VP, CM, and MP.

3. *Define the alphabets of VP and CMP. Reduce the specification of the compose process μASVT to the specification of the concurrent process (VP || CMP) by proving the* ***Compose Restriction Condition,***

$$R_{\mu ASVT}(tr1) \Rightarrow R_{\mu ASVT}(tr1 \upharpoonright \alpha\mu ASVT).$$

The alphabets of the μASVT component processes are defined as

Definitions 17:

αVP = {VPCtl.c, CMCtl.c, RedChan.m, VoiceComsec.m,
VoiceModem.m | c ∈ {cipher, plain}, m ∈ M},

αCM = {CMCtl.c, MPCtl.c, VoiceComsec.m,
ModemComsec.m | c ∈ {cipher, plain}, m ∈ M},and

αMP = {MPCtl.c, BlackChan.m, ModemComsec.m,
VoiceModem.m | c ∈ {cipher, plain}, m ∈ M}.

CM is notified of changes in the control panel by VP through the CMCtl channel. Likewise, MP is notified of changes in the control panel by CM through the MPCtl channel. The alphabet of CMP follows from Definition 9 and Definitions 17,

Lemma 2: αCMP = (αCM ÷ αMP).

The CSP process architecture for the μASVT is shown in Figure 4.

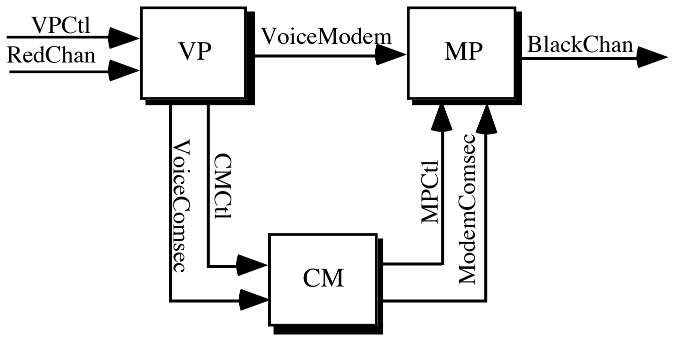

Figure 4. μASVT CSP Architecture

Proof of the Compose Restriction Condition requires showing that the truth of $R_{\mu ASVT}$ depends only on the occurrence of events in μASVT's alphabet. Towards this goal, let ≤ be the trace prefix relation such that, for example, *tr2 ≤ tr1* states that *tr2* is a prefix of *tr1*. The following lemma states sufficient conditions for proving that the value returned by Filter depends only on some restricted set of events:

Lemma 3: $((\text{set1} \subseteq \text{set2}$
$\quad\wedge (\forall\, \text{tr2}.\ \text{tr2} \leq \text{tr1} \Rightarrow \text{req1}(\text{tr2}) = \text{req1}(\text{tr2} \upharpoonright \text{set2})))$
$\quad\Rightarrow \text{Filter}(\text{tr1} \upharpoonright \text{set2},\text{set1},\text{req1}) = \text{Filter}(\text{tr1},\text{set1},\text{req1}))$

Proof: {by mathematical induction on the length of tr1}
Base Case: {definition of $\upharpoonright$}
$\text{Filter}(\langle\,\rangle \upharpoonright \text{set2}, \text{set1}, \text{req1}) = \text{Filter}(\langle\,\rangle, \text{set1}, \text{req1}))$
Induction Step:
$\text{tr1} \upharpoonright \langle\,\rangle \wedge \text{set1} \subseteq \text{set2}$
$\quad\wedge (\forall\, \text{tr2}.\ \text{tr2} \leq \text{tr1} \Rightarrow (\text{req1}(\text{tr2}) = \text{req1}(\text{tr2} \upharpoonright \text{set2})))$
$\quad\wedge (\forall\, \text{tr2}.\ \text{tr2} \leq \text{NonLast}(\text{tr1}) \Rightarrow (\text{req1}(\text{tr2}) = \text{req1}(\text{tr2} \upharpoonright \text{set2})))$
$\Rightarrow \text{Filter}(\text{NonLast}(\text{tr1}) \upharpoonright \text{set2}, \text{set1}, \text{req1})$
$\qquad = \text{Filter}(\text{NonLast}(\text{tr1}), \text{set1}, \text{req1}))$
$\rightarrow$ **{Calculus}**
$\text{tr1} \upharpoonright \langle\,\rangle \wedge \text{set1} \subseteq \text{set2}$
$\wedge (\forall\, \text{tr2}.\ \text{tr2} \leq \text{NonLast}(\text{tr1}) \Rightarrow \text{req1}(\text{tr2}) = \text{req1}(\text{tr2} \upharpoonright \text{set2}))$
$\wedge\ \text{Filter}(\text{NonLast}(\text{tr1}) \upharpoonright \text{set2}, \text{set1}, \text{req1})$
$\qquad = \text{Filter}(\text{NonLast}(\text{tr1}), \text{set1}, \text{req1})$
Case: Last(tr1) $\in$ set2 {definition of $\upharpoonright$}
$(\text{tr1} \upharpoonright \langle\,\rangle \wedge \text{req1}(\text{NonLast}(\text{tr1})) = \text{req1}(\text{NonLast}(\text{tr1}) \upharpoonright \text{set2})$
$\quad\wedge\ \text{Filter}(\text{NonLast}(\text{tr1} \upharpoonright \text{set2}), \text{set1}, \text{req1})$
$\qquad = \text{Filter}(\text{NonLast}(\text{tr1}), \text{set1}, \text{req1}))$
$\rightarrow$ **{Definition 14 of Filter}**
$\text{Filter}(\text{tr1} \upharpoonright \text{set2}, \text{set1}, \text{req1}) = \text{Filter}(\text{tr1}, \text{set1}, \text{req1})$
Case: Last(tr1) $\notin$ set2 {definition of $\upharpoonright$}
$\text{tr1} \upharpoonright \langle\,\rangle$
$\wedge\ \text{Filter}(\text{tr1} \upharpoonright \text{set2},\text{set1},\text{req1}) = \text{Filter}(\text{NonLast}(\text{tr1}),\text{set1},\text{req1})$
$\rightarrow$ **{Definition 14 of Filter}**
$\text{Filter}(\text{tr1} \upharpoonright \text{set2}, \text{set1}, \text{req1}) = \text{Filter}(\text{tr1}, \text{set1}, \text{req1})$
End of Proof.

Since the only place that *tr1* is referenced in Specification 1 of $R_{\mu ASVT}$ is as a parameter to Filter, the Compose Restriction Condition follows from Theorem 1 below. We state and prove this theorem after proving a useful lemma.

Lemma 4: $\{\text{ctlch1.cipher}, \text{ctlch1.plain}\} \subseteq \text{set1}$
$\quad\Rightarrow \text{CipherMode}(\text{ctlch1})(\text{tr1} \upharpoonright \text{set1})$
$\qquad = \text{CipherMode}(\text{ctlch1})(\text{tr1})$

Proof {mathematical induction on the length of tr1}
Base Case: {definition of $\upharpoonright$}
$\text{CipherMode}(\text{ctlch1})(\langle\,\rangle \upharpoonright \text{set1}) = \text{CipherMode}(\text{ctlch1})(\langle\,\rangle)$
Induction Step:
$\text{tr1} \upharpoonright \langle\,\rangle \wedge \{\text{ctlch1.cipher}, \text{ctlch1.plain}\} \subseteq 1$
$\wedge\ \text{CipherMode}(\text{ctlch1})(\text{NonLast}(\text{tr1}) \upharpoonright \text{set1})$
$\qquad = \text{CipherMode}(\text{ctlch1})(\text{NonLast}(\text{tr1}))$

Case: Last(tr1) $\in$ {ctlch1.cipher, ctlch1.plain}
{Definition 15 of CipherMode}
CipherMode(ctlch1)(tr1 $\upharpoonright$ set1) = CipherMode(ctlch1)(tr1)
Case: Last(tr1) $\in$ set1
$\wedge$ **Last(tr1) $\notin$ {ctlch1.cipher,ctlch1.plain}**
{definition of $\upharpoonright$}
tr1 $\upharpoonright$ < > $\wedge$ (CipherMode(ctlch1)(NonLast(tr1 $\upharpoonright$ set1))
= CipherMode(ctlch1)(NonLast(tr1)))
= {Definition 15 of CipherMode}
CipherMode(ctlch1)(tr1 $\upharpoonright$ set1) = CipherMode(ctlch1)(tr1)
Case: Last(tr1) $\notin$ set1
$\wedge$ **Last(tr1) $\notin$ {ctlch1.cipher,ctlch1.plain}**
{definition of $\upharpoonright$}
tr1 $\upharpoonright$ < > $\wedge$ (CipherMode(ctlch1)(tr1 $\upharpoonright$ set1)
= CipherMode(ctlch1)(NonLast(tr1)))
= {Definition 15 of CipherMode}
CipherMode(ctlch1)(tr1 $\upharpoonright$ set1) = CipherMode(ctlch1)(tr1)
End of Proof.

Theorem 1:
(Filter(tr1, BlackChan:M, CipherMode(VPCtl))
= Filter(tr1 $\upharpoonright$ $\alpha\mu$ASVT, BlackChan:M, CipherMode(VPCtl)))
$\wedge$ (Filter(tr1,RedChan:M, CipherMode(VPCtl))
= Filter(tr1 $\upharpoonright$ $\alpha\mu$ASVT, RedChan:M, CipherMode(VPCtl)))

Proof: We prove the first conjunct; the proof of the second conjunct proceeds similarly. By Lemma 3,
(BlackChan:M $\subseteq$ $\alpha\mu$ASVT
$\wedge$ ($\forall$ tr2. tr2 $\leq$ tr1 $\Rightarrow$ (CipherMode(VPCtl)(tr2)
= CipherMode(VPCtl)(tr2 $\upharpoonright$ $\alpha\mu$ASVT))))
$\Rightarrow$ Filter(tr1 $\upharpoonright$ $\alpha\mu$ASVT, BlackChan:M, CipherMode(VPCtl))
= Filter(tr1, BlackChan:M, CipherMode(VPCtl))
$\rightarrow$ **{Definition 13 of $\alpha\mu$ASVT, Lemma 4}**
Filter(tr1 $\upharpoonright$ $\alpha\mu$ASVT, BlackChan:M, CipherMode(VPCtl))
= Filter(tr1, BlackChan:M, CipherMode(VPCtl))
End of Proof.

4. *Derive requirements R_{VP} for VP and R_{CMP} for CMP with the goal of proving VP || CMP satisfies $R_{\mu ASVT}$.*

$R_{\mu ASVT}$ states that every message transmitted by the terminal when in the cipher mode is a proper transformation of some message received by the terminal. A natural requirement decomposition is to require that, when in the cipher mode, every message transmitted by a component be a proper transformation of some message received by the component. A proper transformation for VP is to Analyze the message; a proper transformation for CMP is to Encrypt, using the encryption key Key, and then Modulate the message. This suggests the following specifications for VP and CMP, respectively:

Specification 2: VP sat R_{VP}

R_{VP}(tr1) = ∀ m1. (VoiceComsec.m1 **in** Filter(tr1, VoiceComsec:M, CipherMode(VPCtl))
⇒ ∃ m2.(RedChan.m2 **in** Filter(tr1, RedChan:M, CipherMode(VPCtl))
∧ Analyze(m2) = m1))

Specification 3: CMP sat R_{CMP}

R_{CMP}(tr1) = ∀ m1.(BlackChan.m1 **in** Filter(tr1, BlackChan:M,CipherMode(CMCtl))
⇒ ∃m2.(VoiceComsec.m2 **in** Filter(tr1,VoiceComsec:M, CipherMode(CMCtl))
∧ Modulate(Encrypt(m2,Key)) = m1))

Notice that communications over VPCtl are used to determine whether the terminal is in the cipher mode for VP, whereas communications over CMCtl are used for CMP.

5. *Prove the* ***Concurrent Restriction Condition,***

R_{VP} (tr1 ↾ αVP) ⇒ R_{VP} (tr1) ∧ R_{CMP}(tr1 ↾ αCMP) ⇒ R_{CMP}(tr1).

If this proof fails, revise the requirements so that R_{VP} depends only on the events in the alphabet of VP and that R_{CMP} depends only on events in the alphabet of CMP and try again.

We prove the first conjunct of the Concurrent Restriction Condition; the proof of the second conjunct proceeds similarly. Since the only place that *tr1* is referenced in Specification 2 of VP is as a parameter to Filter, the first conjunct of the Concurrent Restriction Condition follows from Theorem 2 below. This proof has the same general structure as the proof of Theorem 1.

Theorem 2:
(Filter(tr1, VoiceComsec:M, CipherMode(VPCtl))
= Filter(tr1 ↾ αVP, VoiceComsec:M, CipherMode(VPCtl)))
∧ (Filter(tr1,RedChan:M, CipherMode(VPCtl))
= Filter(tr1 ↾ αVP, RedChan:M,CipherMode(VPCtl)))

Proof: We prove the first conjunct; the proof of the second conjunct proceeds similarly. By Lemma 3,
(VoiceComsec:M ⊆ αVP
∧ (∀ tr2. tr2 ≤ tr1 ⇒ (CipherMode(VPCtl)(tr2)
= CipherMode(VPCtl)(tr2 ↾ αVP))))
⇒ Filter(tr1 ↾ αVP, VoiceComsec:M, CipherMode(VPCtl))
= Filter(tr1, VoiceComsec:M, CipherMode(VPCtl))
→ **{Lemma 4, Definitions 17 of αVP}**
Filter(tr1 ↾ αVP, VoiceComsec:M,CipherMode(VPCtl))
= Filter(tr1, VoiceComsec:M,CipherMode(VPCtl))
End of Proof.

6. *Attempt to prove the* ***Conjunction Condition,***

(R_{VP} (tr1) ∧ R_{CMP}(tr1)) ⇒ $R_{\mu ASVT}$(tr1).

If successful, continue the tree traversal to the next vertex at step 3. Otherwise, specify the weakest condition, C, needed to complete the proof.

This step is difficult, partly due to the fact that the proof is dependent on the proper synchronization of VP and CMP. We derive the necessary synchronization requirement through an attempt to prove the Conjunction Condition. Analyzing R_{VP} and R_{CMP} of Specifications 2 and 3 shows that the Conjunction Condition can be deduced from the following theorem:

Theorem 3: (Filter(tr1, VoiceComsec:M, CipherMode(VPCtl))
  = Filter(tr1,VoiceComsec:M, CipherMode(CMCtl))
 ∧ Filter(tr1, BlackChan:M, CipherMode(VPCtl))
  = Filter(tr1, BlackChan:M, CipherMode(CMCtl)))
 ⇒ ((R_{VP}(tr1) ∧ R_{CMP}(tr1)) ⇒ $R_{\mu ASVT}$(tr1))

Proof:
Filter(tr1, VoiceComsec:M, CipherMode(VPCtl))
  = Filter(tr1,VoiceComsec:M, CipherMode(CMCtl))
 ∧ Filter(tr1, BlackChan:M, CipherMode(VPCtl))
  = Filter(tr1, BlackChan:M, CipherMode(CMCtl))
 ∧ R_{VP}(tr1) ∧ R_{CMP}(tr1)
→ **{Specification 2 of R_{VP} and Specification 3 of R_{CMP}}**
∀ m1.(VoiceComsec.m1 **in** Filter(tr1, VoiceComsec:M,
  CipherMode(VPCtl))
 ⇒ ∃ m2.(RedChan.m2 **in** Filter(tr1, RedChan:M,
  CipherMode(VPCtl))
` ∧ Analyze(m2) = m1))
∧ ∀ m1.((BlackChan.m1 **in** Filter(tr1, BlackChan:M,
  CipherMode(VPCtl))
 ⇒ ∃ m2.(VoiceComsec.m2 **in** Filter(tr1, VoiceComsec:M,
  CipherMode(VPCtl))
 ∧ Modulate(Encrypt(m2, Key)) = m1))
→ **{Calculus}**
∀ m1.(BlackChan.m1 **in** Filter(tr1, BlackChan:M,
  CipherMode(VPCtl))
 ⇒ ∃ m2.(RedChan.m2 **in** Filter(tr1, RedChan:M,
  CipherMode(VPCtl))
 ∧ Modulate(Encrypt(Analyze(m2), Key)) = m1))
= **{Specification 1 of $R_{\mu ASVT}$}**
$R_{\mu ASVT}$(tr1)
End of Proof.

Theorem 3 states sufficient conditions for proving the Conjunction Condition. Provided these conditions are true, when the terminal is in the cipher mode, the origination of a message transmitted through BlackChan can be traced through CMP from VoiceComsec. Likewise, the origination of a message transmitted through VoiceComsec can be traced through VP from RedChan. Unfortunately we cannot prove the antecedent of Theorem 3 without additional information about the valid traces of the μASVT. Therefore, set C equal to this antecedent.[11]

7. *Describe C as a conjunction of simple conditions in conjunctive normal form. If no conjunct depends solely on the traces of either VP or CMP then conjoin C to $SR_{\mu ASVT}$ and continue the tree traversal to the next vertex at step 3. Otherwise, conjoin to R_P each conjunct of C that depends only on the traces of P (for P = VP or P = CMP) and continue at step 5.*

By mathematical induction on *tr1* and Definition 14 of Filter, C reduces to the conjunction of two simpler expressions:

Condition C:

$$\begin{aligned}&((tr1 \neq \langle\,\rangle \wedge \mathrm{Last}(tr1) \in \mathrm{VoiceComsec{:}M})\\ &\quad \Rightarrow (\mathrm{CipherMode(VPCtl)}(tr1) = \mathrm{CipherMode(CMCtl)}(tr1)))\\ &\wedge ((tr1 \neq \langle\,\rangle \wedge \mathrm{Last}(tr1) \in \mathrm{BlackChan{:}M})\\ &\quad \Rightarrow (\mathrm{CipherMode(VPCtl)}(tr1) = \mathrm{CipherMode(CMCtl)}(tr1))).\end{aligned}$$

Recall that CipherMode(*ctlch1*)(*tr1)* is true if and only if the terminal is in the cipher mode after engaging in the sequence of events *tr1*. The mode is cipher if and only if either the most recent transmission over *ctlch1* was cipher or, if there is no such transmission, the initial mode is cipher. Since VPCtl and CMCtl are distinct channels, there is no way to determine the truth of C from only the definition of CipherMode.

The truth of the first conjunct of C depends only on communication over VoiceComsec, VPCtl, and CMCtl; the truth of the second conjunct depends on communication over BlackChan, VPCtl, and CMCtl. Since VP must participate in any events of *tr1* that occur in its alphabet and since, by Definitions 17 of αVP,

Lemma 5: $\{\mathrm{VoiceComsec}.m, \mathrm{VPCtl}.c, \mathrm{CMCtl}.c \mid c \in \{\mathrm{cipher}, \mathrm{plain}\}, m \in M\} \subseteq \alpha\mathrm{VP}$,

the truth of the first conjunct depends solely on the traces of VP. Defining R'_{VP} as this conjunct results in

Definition 18:

$$\begin{aligned}R'_{VP}(tr1) = &((tr1 \neq \langle\,\rangle \wedge \mathrm{Last}(tr1) \in \mathrm{VoiceComsec{:}M})\\ &\Rightarrow (\mathrm{CipherMode(VPCtl)}(tr1)\\ &\qquad = \mathrm{CipherMode(CMCtl)}(tr1)))\end{aligned}$$

Therefore, we can conjoin R'_{VP} to R_{VP}. Unfortunately, the second conjunct of C does not depend on the traces of a single component. By Definitions 17, communications over BlackChan and VPCtl do not occur in the alphabet of any single component. Nevertheless, since the specification of VP has been extended, we must go back to step 5 and reprove the Concurrent Restriction Condition for VP.

D. Justifying the Revised Decomposition

During the first attempt to decompose the requirement of μASVT, we proved the Concurrent Restriction Condition for the original definitions of R_{CMP} and R_{VP}. Conjoining

[11] Note that since this instantiation of C is not necessary to establish the truth of the Conjunction Condition, we cannot guarantee that the set of synchronization requirements derived is minimal.

R'_{VP} to R_{VP}, requires proof that R'_{VP} depends only on the events in the alphabet of VP to re-establish the truth of the Concurrent Restriction Condition:

Theorem 4: $R'_{VP}(tr1 \upharpoonright \alpha VP) \Rightarrow R'_{VP}(tr1)$

Proof:

$R'_{VP}(tr1 \upharpoonright \alpha VP)$

= {Definition 18 of R'_{VP}}

$(tr1 \upharpoonright \alpha VP \neq < > \wedge \text{Last}(tr1 \upharpoonright \alpha VP) \in \text{VoiceComsec:M})$
$\Rightarrow (\text{CipherMode(VPCtl)}(tr1 \upharpoonright \alpha VP)$
$= \text{CipherMode(CMCtl)}(tr1 \upharpoonright \alpha VP)))$

→ {Definitions 17 of αVP}

$(tr1 \neq < > \wedge \text{Last}(tr1) \in \text{VoiceComsec:M})$
$\Rightarrow (\text{CipherMode(VPCtl)}(tr1 \upharpoonright \alpha VP)$
$= \text{CipherMode(CMCtl)}(tr1 \upharpoonright \alpha VP)))$

→ {Lemma 4, Definitions 17 of αVP}

$(tr1 \neq < > \wedge \text{Last}(tr1) \in \text{VoiceComsec:M})$
$\Rightarrow (\text{CipherMode(VPCtl)}(tr1) = \text{CipherMode(CMCtl)}(tr1))$

= {Definition 18 of R'_{VP}}

$R'_{VP}(tr1)$

End of Proof.

Since R_{VP} has been modified, the next step is to reprove the Conjunction Condition. Fortunately, these proofs proceed exactly as before except, this time, the proof depends only on the second conjunct of Condition 1. Since it does not depend on any one component of the μASVT, the decomposition at this vertex is finished.

E. Analyzing the Final Results

Decomposing Specification 1 of μASVT results in a specification for each component and a number of synchronization requirements on multiple components. In summary, the derived component specifications are as follows:

Specification 4: VP sat R_{VP}

$R_{VP}(tr1) = (\forall\, m1.\ (\text{VoiceComsec}.m1\ \textbf{in}\ \text{Filter}(tr1, \text{VoiceComsec:M}, \text{CipherMode(VPCtl)})$
$\Rightarrow \exists\, m2.\ (\text{RedChan}.m2\ \textbf{in}\ \text{Filter}(tr1, \text{RedChan:M}, \text{CipherMode(VPCtl)})$
$\wedge\ \text{Analyze}(m2) = m1))$
$\wedge\ (tr1 \neq < > \wedge \text{Last}(tr1) \in \text{VoiceComsec:M})$
$\Rightarrow (\text{CipherMode(VPCtl)}(tr1) = \text{CipherMode(CMCtl)}(tr1))$

Specification 5: CM sat R_{CM}

$R_{CM}(tr1) = (\forall m1.(\text{ModemComsec}.m1\ \textbf{in}\ \text{Filter}(tr1,\text{ModemComsec:M}, \text{CipherMode(CMCtl)})$
$\Rightarrow \exists\, m2.\ (\text{VoiceComsec}.m2\ \textbf{in}\ \text{Filter}(tr1, \text{VoiceComsec:M}, \text{CipherMode(CMCtl)})$
$\wedge\ \text{Encrypt}(m2, \text{Key}) = m1))$
$\wedge\ (tr1 \neq < > \wedge \text{Last}(tr1) \in \text{ModemComsec:M})$
$\Rightarrow (\text{CipherMode(CMCtl)}(tr1) = \text{CipherMode(MPCtl)}(tr1))$

Specification 6: MP sat R_{MP}

$R_{MP}(tr1) = \forall$ m1. (BlackChan.m1 **in** Filter(tr1, BlackChan:M, CipherMode(MPCtl))
$\Rightarrow \exists$m2.(ModemComsec.m2 **in** Filter(tr1, ModemComsec:M, CipherMode(MPCtl))
$\wedge$ Modulate(m2) = m1)).

The first conjunct of R_{VP} in Specification 4, the first conjunct of R_{CM} in Specification 5, and all of R_{MP} in Specification 6 represent the proper transformation of messages transmitted through each component when in the cipher mode. Each processor uses its own control channel for determining whether the terminal is in the cipher mode. At the system level, however, only VPCtl is used to determine whether the system is in the cipher mode. As suggested by the analysis of Sections IV-C and IV-D this fact requires that VP and MP satisfy additional conditions, defined by the second conjuncts of R_{VP} and R_{CM} in Specifications 4 and 5. These conditions were derived during the effort to prove the Conjunction Conditions during the first and second level decompositions.

The second conjuncts of R_{VP} and R_{CM} are somewhat obscure. Their purpose is to help ensure that the position of the mode selector dial is seen consistently by all three processes. A notification of a change in mode for the μASVT requires the synchronized, and sequential, notification of each of the three component processors over the control channels, VPCtl, CMCtl, and MPCtl. The second conjuncts form an integral part of the requirement that the notifications proceed uninterrupted by message transmissions. The second conjunct of R_{VP} states that every mode change received by VP over VPCtl be sent to CM over CMCtl before another message is accepted for transmission. Since this is a requirement of VP it is stated as a part of R_{VP}. Likewise, the second conjunct of R_{CM} states that every value received by CM over CMCtl be sent to MP over MPCtl before another message is accepted for transmission. Since this is a requirement of CM it is stated as a part of R_{CM}. MP has no such responsibility.

Specifications 4 and 5 are not sufficient to ensure that a mode change is not interrupted. Unfortunately, this cannot be enforced merely by specifying requirements on the component processes in isolation; two requirements on multiple components are needed:

Requirement 1:
(ValidTrace(tr1, μASVT) $\wedge$ tr1 $\neq$ < > $\wedge$ Last(tr1) $\in$ BlackChan:M)
$\Rightarrow$ (CipherMode(VPCtl)(tr1) = CipherMode(CMCtl)(tr1))

Requirement 2:
(ValidTrace(tr1, CMP) $\wedge$ tr1 $\neq$ < > $\wedge$ Last(tr1) $\in$ BlackChan:M})
$\Rightarrow$ (CipherMode(CMCtl)(tr1) = CipherMode(MPCtl)(tr1)).

These conditions are very similar to the second conjuncts of R_{VP} and R_{CM}. Intuitively, they state that whenever a message is transmitted by BlackChan, the current mode of the terminal is cipher if and only if the most recent values transmitted over VPCtl, MPCtl, and CMCtl were cipher, or, if no values were transmitted over these channels, the terminal began in cipher mode. The component alphabets given by Definitions 17 imply that neither of these conditions is the sole responsibility of a single component of the μASVT. Requirements 1 and 2 form the set of synchronization requirements of the μASVT as referenced in the introduction of this paper.

Although it is outside the scope of this paper to prove the requirements described in this section, it is interesting to look at them in light of the example functional refinement presented in the appendix to this paper. Through some analysis of these definitions, the truth of Specifications 4 through 6, and Requirements 1 and 2, becomes apparent. Proofs of the component requirements are relatively easy to formalize using the proof methods of the Trace Model of CSP. Unfortunately, however, when one tries to formalize the proofs of Requirements 1 and 2, the arguments explode into an extremely large number of cases, even for this relatively simple example. Future work requires determining methods for practically dealing with such proofs.

V. Summary and Conclusions

This paper describes and demonstrates a method for formalizing and decomposing critical requirements of systems using the Trace Model of CSP. The method proceeds iteratively, until the appropriate requirements for the component processes and the minimal set of synchronization requirements are found. An extension to the CSP notation, involving process composition with hidden internal structure, promotes hierarchical system design and decomposition. A goal of the decomposition process is to minimize the number and complexity of the synchronization requirements since these are the most difficult to verify in later system development. We capture these requirements in the simplest possible context and ensure that each depends on the behavior of at least two component processes. The method described reduces the problem of assuring that the system meets its critical requirements to the simpler, although possibly nontrivial, problem that each component meets its derived requirements and the system satisfies the derived synchronization requirements.

Experience with the μASVT decomposition confirms our belief that the CSP Trace Model is a valuable formalism for specifying and reasoning about systems and their requirements. Although the Trace Model is restricted to reasoning about the safety of non-divergent processes, within this domain the power of trace theory eases the description of critical properties over less sophisticated models while avoiding the complexities of the more sophisticated models. For example, the use of separate channel histories, as in the Counter Model and Gypsy, complicates the statement of relationships between the communication histories of different channels at specific times during a process's execution. The μASVT's Red/Black separation requirement is difficult, if not impossible, to specify within this paradigm due to the need to determine the mode in which the terminal is operating when messages are transmitted; this determination requires finding out the last value transmitted over VPCtl for each message transmitted over BlackChan. The Trace Model allows the statement of such properties by describing process behavior in terms of interleavings of distinct channel histories. The ability to decompose critical liveness properties of potentially diverging processes using the more sophisticated models is an area of future research.

Stages of development, subsequent to system decomposition, require methods to verify sequential component and synchronization requirements. Verifying properties of sequential components does not present any inherent difficulties. Unfortunately, we have not yet found a practical approach applicable to nontrivial systems for formally proving the derived synchronization requirements. Although proof methods for reasoning about global properties exist within the CSP Trace Model, they are intractable for most real-world systems. This problem is not unique to the model used here; the other models of CSP discussed, as well as the proof methods of Gypsy, exhibit these same difficulties. In general, if it is impossible to derive component process requirements that are sufficient to prove the system requirements, reasoning about global properties is required. This condition arose in the decomposition of even the relatively simple μASVT example as discussed in section IV.

The EHDM verification system [8] is proving useful for mechanically checking the proofs required of the decomposition process. We have encoded within the logic of EHDM the relevant portion of the CSP Trace Model allowing system architecture and requirements to be specified. Directed by an informal proof outline derived separately, we can guide EHDM through the seven-step method. When applied to the μASVT decomposition, EHDM supplied many of the low-level proof details, e.g., variable instantiations, and, at times, caught errors in the informal proof. The final EHDM proof provided necessary information for revising and refining the informal proofs. This paper presents a subset of these revised proofs.

Although the use of EHDM is beneficial towards checking the integrity of a decomposition, it is clear that the availability of tools specifically oriented towards specifying and verifying systems involving concurrency are needed to facilitate the decomposition process. Verification systems with much of the concurrency model built in would allow the automation of large portions of the proof process. Alternatively, verification systems that allow the user to define automatically invoked decision procedures would permit developers to "program" the theorem prover with CSP-specific proof strategies. Experiences such as those described in this paper may prove beneficial for identifying useful characteristics of verification tools and helpful decision procedures for verifying concurrent systems.

The decomposition method proposed in this paper applies to systems with critical requirements implemented in either hardware or combinations of software and hardware. Executable languages supporting concurrency [43,44,45] may be useful for prototyping systems described in CSP [46,47,48], for testing the validity of formalisms specified in the Trace Model [49], and for continuing the formal verification to lower levels of system design and implementation [43]. Others are investigating the incorporation of CSP into formal methods that do not explicitly deal with concurrency such as the Z specification language [4]. If decompositions are taken to a sequential level, more conventional procedural languages may be used to implement systems in software or as design languages for hardware. Formal reasoning can proceed in such languages if a trace semantics can derived for programs written, e.g., [39]. Since our method focuses on requirement decomposition rather than physical decomposition, it can be used during structural hardware design to decompose requirements for hardware in a top-down fashion as primitive hardware elements are interconnected to realize higher-level blocks in a bottom-up fashion.

Decomposing requirements using formal methods enhances the role of testing and simulation in the design of trustworthy systems. A designer tests a system description to determine whether it meets the requirements. Complex systems lead to complex test suites. The greater the complexity of the test suite, the more difficult it is to determine whether the system meets its requirements. A requirement decomposition allows the designer to break the test suite for a system into smaller, less complex test suites for its components. This simplifies the requirements that must be assured and decreases the number of execution paths that must be traversed. Simplifying the testing process in this way increases the probability of finding problems that exist in the design. Of course, each synchronization requirement derived from a requirement decomposition requires testing of those components on which it depends.

Further work is proceeding along two complementary lines: the application of the decomposition method to a security-critical portion of a full-scale communication system and the extension of the method to handle the verification of the component-level and synchronization requirements. From derived component-level requirements, we expect to generate and verify implementations of the components using some formally verifiable programming language such as Gypsy [29] or m-Verdi [50]. Unfortunately, due to the immature state of formally verifiable languages, the actual source code will likely be written in a more conventional language with better run-time support. Nevertheless, the verified implementations will act as low-level functional descriptions from which to derive the source code. Using strict coding conventions and review procedures this derivation should be fairly straightforward so as to preserve the formal verification accomplished at the higher levels.

Acknowledgement

The author is grateful to Carl Landwehr, John McLean, Charles Payne, and James Gray of NRL; to Ken Hayman of the Australian DSTO; and to the referees, for helpful comments on earlier drafts of this paper.

References

[1] J. Jacob, "Security Specifications" *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, pp. 14-23, April 1988.

[2] A.P. Moore, "Investigating Formal Specification and Verification Techniques for Comsec Software Security," in *Proceedings of the 11th National Computer Security Conference*, pp. 129-138, Oct. 1988.

[3] C.T. Sennett, "Tool Support for the Production of High Integrity Software," Royal Signals Radar Establishment, U. K., Report No. 89005, April 1989.

[4] J.C.P. Woodcock, "Transaction Processing Primitives and CSP," *IBM Journal of Research and Development*, Vol. 31, No. 5, September 1987.

[5] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[6] E.R. Olderog and C.A.R. Hoare, "Specification Oriented Semantics for Communicating Sequential Processes," *ACTA Informatica* 23, pp. 9-66, 1986.

[7] Hoare, C.A.R., "A Model For Communicating Sequential Processes, in *On the Construction of Programs* (R:M. McKeag and A:M. McNaughton, eds.) Cambridge University Press, pp. 229-243, 1980.

[8] "EHDM Specification and Verification System Version 4.1, User's Guide," Computer Science Laboratory, SRI International, May 1988.

[9] "EHDM Specification and Verification System: Preliminary Definition of the EHDM Specification Language," Computer Science Laboratory, SRI International, Jan. 1990.

[10] J. van de Snepscheut, "Trace Theory and VLSI Design," *Lecture Notes in Computer Science 200*, Springer-Verlag, Berlin Heidelberg, 1985.

[11] A.P. Moore, "The Specification and Verified Decomposition of an Example Communication System," Naval Research Laboratory Technical Memorandum 5540-019:APM:apm, Jan. 1990.

[12] A.P. Moore, "A Method for Decomposing Requirements of Systems of Communicating Components," Naval Research Laboratory Technical Memorandum 5540-309:APM:apm, Sep. 1989.

[13] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, pp. 666-677, 1978.

[14] R. Milner, "A Calculus of Communicating Systems," *Lecture Notes in Computer Science 92*, Springer-Verlag, Berlin Heidelberg New York, 1980.

[15] R. Milner, *Communication and Concurrency*, Prentice-Hall International,Englewood Cliffs, New Jersey, 1989.

[16] S.D. Brookes, "On the Relationship of CCS and CSP," *Lecture Notes in Computer Science 154*, 1983.

[17] J. Hooman and W.P. de Roever, "The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP," *Lecture Notes in Computer Science 224*, Springer-Verlag, Berlin Heidelberg, 1985.

[18] H. Barringer, "A Survey of Verification Techniques for Parallel Programs," *Lecture Notes in Computer Science 191*, Springer-Verlag Berlin Heidelberg, 1985.

[19] C.A.R. Hoare, "A Calculus of Total Correctness for Communicating Processes," *Science of Computer Programming 1*, pp. 49-72, 1981.

[20] C.A.R. Hoare, "Specifications, Programs and Implementations," Technical Monograph PRG-29, Programming Research Group, Oxford University, 1982.

[21] J. Misra and K:M. Chandy, "Proofs of Networks of Processes," *IEEE Transactions on Software Engineering*, SE-7, pp. 417-426, 1981.

[22] Z.C. Chen and C.A.R. Hoare, "Partial Correctness of Communicating Sequential Processes," *Proc. International Conference on Distributed Computing*, Paris, April 1981.

[23] N. Francez, D. Lehmann, and A. Pnueli, "A Linear History Semantics for Distributed Programming," *TCS* 32, pp. 25-46, 1984.

[24] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe, "A Theory of Communicating Sequential Processes," *Journal of the Association for Computing Machinery*, Vol. 31, No.3, pp. 560-599, July 1984.

[25] S.D. Brookes and A.W. Roscoe, "An Improved Failures Model for Communicating Sequential Processes," *Lecture Notes in Computer Science 154*, pp. 281-305, 1985.

[26] A.W. Roscoe, "A Semantics and Proof System for Communicating Processes," *Lecture Notes in Computer Science 164*, pp. 68-85, 1984.

[27] C.A.R. Hoare, "Algebraic Specifications and Proofs for Communicating Sequential Processes," NATO ASI Series, Vol. F36, *Logic of Programming and Calculi of Discrete Design* (M. Broy, ed.), Springer-Verlag, Berlin Heidelberg,1987.

[28] D.I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," Institute for Computing Science, The University of Texas at Austin, Tech. Rep. ICSCA-CMP-15, Jan. 1979.

[29] D.I. Good., R. L. Akers, and L. M. Smith, "Report on Gypsy 2.05," Computational Logic, Inc., Tech. Rep. #1-b, Jan. 1989.

[30] L. Lamport, "The "Hoare Logic" of Concurrent Programs," *ACTA Informatica 14*, pp. 21-37, 1980.

[31] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs:An Axiomatic Approach," *Communications of the ACM*, Vol. 19, pp. 279-285, May 1976.

[32] G:M. Levin and D. Gries, "A Proof Technique for Communicating Sequential Processes," *ACTA Informatica 15*, pp. 281-302, 1981.

[33] K.R. Apt, N. Francez and W.P. de Roever, "A Proof System for Communicating Sequential Processes," *ACM Trans. Program. Lang. Syst.*, Vol. 2, No.. 3, pp. 359-385, July 1980.

[34] N. Soundararajan and O.J. Dahl, "Partial Correctness Semantics for Communicating Sequential Processes," Res. Rep. 66, Institute for Informatics, University of Oslo, Norway.

[35] J. Zwiers, W.P. de Roever, and P. van Emde Boas, "Compositionality and Concurrent Networks," *Lecture Notes in Computer Science 194*, pp. 509-519, 1985.

[36] A.W. Bartussek and D.L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules," University of North Carolina, Chapel Hill, North Carolina, Report TR 77-012, Dec. 1977.

[37] D.L. Parnas and Y. Wang, "The Trace Assertion Method of Module Interface Specification," Queen's University, Kingston, Ontario, Technical Report 89-261, October 1989.

[38] J. McLean, "A Formal Foundation for the Abstract Specification of Software," *Journal of the Association for Computing Machinery* 31(3), pp. 600-627, July 1984.

[39] J. McLean, "Using Trace Specifications for Program Semantics and Verification," Naval Research Laboratory Report 9033, April 1987.

[40] D. McCullough, "Noninterference and the Composability of Security Properties," in *Proceedings 1988 Symposium on Security and Privacy*, Oakland, CA., IEEE Computer Society, Apr. 1988.

[41] J.C.P. Woodcock, "Transaction Processing Primitives and CSP," *IBM Technical Journal* 31 5, pp. 535-545, Sep. 1987.

[42] J. Jacob, "On the Derivation of Secure Components," in *Proceeding 1989 Symposium on Security and Privacy*, Oakland, California, May 1989.

[43] INMOS Ltd, *Communicating Process Architecture*, Prentice Hall International (UK), 1988.

[44] P.B. Hansen, "The Joyce Language Report," *Software Practice and Experience (UK)*, Vol. 19, No. 6, pp. 579-592, June 1989.

[45] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, (eds.), *The Formal Description Technique LOTOS*, Elsevier-North Holland, 1989.

[46] G.V. Collis and E.J. Kappos, "OCCAM as a Hardware Description Language," *Software Engineering Journal*, pp.213-219, Nov. 1987.

[47] B.J. Curry. "Language based architecture eases system design - III," *Computer Design* 23(1), 127-136, January 1984.

[48] R.D. Dowsing, "Simulating Hardware Structures in OCCAM," *Software & Microsystems*, Vol. 4, No. 4, pp. 77-84, August 1985.

[49] A.W. Roscoe, "Denotational Semantics for Occam," in *Proceedings of the NSF/SERC Workshop on Concurrency, Lecture Notes in Computer Science 197*, Springer-Verlag, July 1984.

[50] D. Craigen, "A Description of m-Verdi," I.P. Sharp Associates Limited Technical Report TR-87-5420-02, August 1985.

Appendix: A CSP Implementation of the μASVT

The following functional description reflects the operation of the terminal as discussed in Section IV. Note that the CSP recursive process μX.F(X) represents the process X such that X = F(X). The choice process (*e1* → *P*) | (*e2* → *Q*) represents a process that first engages in either *e1* or *e2*. If it engages in *e1* then it, subsequently, behaves like *P*; if it engages in *e2* then it, subsequently, behaves like *Q*.

```
VP = VPCtl ? mode
      → μY.(CMCtl ! mode
             → if mode = cipher
                then μX. ((RedChan ? msg  → VoiceComsec ! Analyze(msg) → X)
                         | VPCtl ? mode → Y)
                else μX. ((RedChan ? msg → VoiceModem ! Analyze(msg) → X)
                          | VPCtl ? mode → Y))

CM = CMCtl ? mode →
      → μY.(MPCtl ! mode
             → if mode = cipher
                then μX. ((VoiceComsec? msg → ModemComsec ! Encrypt(msg, Key) → X)
                          | CMCtl ? mode → Y)
                else CMCtl ? mode → Y)

MP = MPCtl ? mode
      → μY.(if mode = cipher
           then μX. ((ModemComsec? msg  → BlackChan ! Modulate(msg) → X
                     | MPCtl ? mode → Y))
           else μX. ((VoiceModem ? msg → BlackChan ! Modulate(msg) → X)
                    | MPCtl ? mode → Y)
```